Ph.D. DISSERTATION

Optimizing Machine Learning Inference Systems with White Box Approaches

White Box 방식을 통한 머신러닝 추론 시스템 최적화

FEBRUARY 2020

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Yunseong Lee

Ph.D. DISSERTATION

Optimizing Machine Learning Inference Systems with White Box Approaches

White Box 방식을 통한 머신러닝 추론 시스템 최적화

FEBRUARY 2020

DEPARTMENT OF COMPUTER SCIENCE ENGINEERING
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Yunseong Lee

Optimizing Machine Learning Inference Systems with White Box Approaches

White Box 방식을 통한 머신러닝 추론 시스템 최적화

지도교수 전병곤

이 논문을 공학박사학위논문으로 제출함

2019 년 11 월

서울대학교 대학원

컴퓨터 공학부

이윤성

이윤성의 박사학위논문을 인준함

2019 년 12 월

위 원 장	이 재 욱	(인)
부위원장	전 병 곤	(인)
위 원	강 유	(인)
위 원	이 영 기	(인)
위 원	전 명 재	(인)

Abstract

Optimizing Machine Learning Inference Systems with White Box Approaches

Yunseong Lee
School of Computer Science Engineering
Collage of Engineering
The Graduate School
Seoul National University

Machine Learning inference systems support user-facing services and have strict performance requirements. Specifically, the systems need to provide low latency, high throughput, while using minimal resources. The simplest way to deploy models is wrapping the models into black boxes such as containers. This approach eases deployment but possible optimizations are limited making its performance sub-optimal as we want to run many models together sharing resources.

In this dissertation, we propose a white box model serving, which enables both end-to-end and multi-model optimizations; models are restructured to an optimized execution plan and resources are shared among the models running together. We introduce PRETZEL, our implementation of the white box approach. Our evaluation with production-scale model pipelines shows that white box optimizations can introduce performance improvements with respect to the latency, memory footprint, and throughput, compared to the state-of-the-art systems in the black box approaches.

Keywords: Machine Learning Inference System, Prediction Serving System, Performance Optimization, White Box Optimization

Student Number: 2014-21771

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, "intelligent" services such as Microsoft Cortana speech recognition, Netflix movie recommender or Gmail spam detector depend on ML scoring capabilities, which are currently experiencing a growing demand [37]. This, in turn fosters the research in inference systems in cloud settings [23, 61, 4, 38], where trained models from data science experts are operationalized.

Machine Learning (ML) is usually conceptualized as a two-steps process: first, during *training* model parameters are estimated from large datasets by running computationally intensive iterative algorithms; successively, trained models are used for *inference*[1] to generate predictions through the estimated model parameters. ML models often are *pipeline*s with operators to massage and featurize the raw input data and ML model for rendering prediction.

When trained pipelines are served for inference, the full set of operators is deployed altogether. However, pipelines have different system characteristics based on the phase in which they are employed: for instance, at training time ML models run complex algorithms to scale over large datasets (e.g., linear models can use gradient descent in one of its many flavors [67, 65, 70]), while,

[1] inference is also referred as prediction serving, model serving, scoring, etc.

once trained, they behave as other regular featurizers and data transformations.

Furthermore, during inference pipelines are often surfaced for direct users' servicing and therefore have the following performance requirements:

- *low latency* (in the order of milliseconds) because scoring is often one segment in more complex services (e.g., smartphone or web applications) that potentially provide a Service Level Agreement (SLA).
- *efficient resource usage* (e.g., memory, CPU) to save operational costs.
- *high throughput* to handle as many concurrent requests as possible
- *graceful performance degradation* in case of load spikes.

Existing prediction serving systems, such as Clipper [4, 38], TensorFlow Serving [23, 61], Rafiki [75], ML.NET [14], and others [18, 20, 56, 15] focus mainly on ease of deployment, where model pipelines are considered as *black boxes* (e.g., containerization). Under this strategy, systems can apply *pipeline-agnostic* optimizations such as handling multiple requests in batches and caching the results of the inference if some predictions are frequently issued for the same pipeline. These techniques assume no knowledge and no control over the pipeline and are unaware of its internal structure.

Nevertheless, we found that black box approaches fell short on several aspects. For instance, prediction services are profitable for ML-as-a-service providers only when pipelines are accessed in batch or frequently enough, and may not be when models are accessed sporadically (e.g., twice a day, a pattern we observed in practice) or not uniformly. Also, increasing model density in machines, thus increasing utilization is not always possible for two reasons: first, higher model density increases the pressure on the memory system, which is sometimes dangerous—we observed (Section 4.4) machines swapping or blocking when too many models are loaded; as a second reason, co-location of models may increase

tail latency especially when seldom-used models are swapped to disk and later re-loaded to serve only a few users' requests.

Inspired by these and other limitations of the existing black box approaches (further described in Section 3.1), we propose a *white box* approach for model serving [54]. Starting from the observation that trained pipelines often share operators and parameters (such as weights and dictionaries used within operators, and especially during featurization [82]), we apply end-to-end and multi-pipeline optimization techniques similar to database systems to reduce resource usage while improving performance.

In PRETZEL, our implementation of white box approaches, deployment and serving of model pipelines follow a two-phase process: During an *off-line phase*, statistics from training and state-of-the-art techniques from in-memory data-intensive systems [40, 84, 32, 51, 60] are used in concert to optimize and compile operators into *model plans*. Model plans are white box representations of input pipelines such that PRETZEL is able to store and re-use parameters and computation among similar plans. In the *on-line phase*, memory (data vectors) and CPU (thread-based execution units) resources are pooled among plans. When an inference request for a plan is received, an event-based scheduling [76] is used to bind computation to execution units.

Using 500 different production-scale pipelines used internally at Microsoft, we prove the impact of the above design choices with respect to ML.NET and end-to-end solutions such as Clipper. Specifically, PRETZEL is on average able to improve memory footprint by 25×, reduce the 99th percentile latency by 5.5×, and increase the throughput by 4.7×.

In summary, our contributions are:

- A thorough analysis of the problems and limitations burdening black box model serving approaches;
- A set of design principles for white box model serving allowing pipelines

to be optimized for inference and to share resources;

- A system implementation of the above principles;
- An experimental evaluation showing order-of-magnitude improvements over several dimensions compared to previous black box approaches.

The remainder of the paper is organized as follows:

Chapter 2 describes the background of ML pipeline and black box prediction serving. We first explain the notion of machine learning pipelines (Section 2.1) and present ML.NET, a ML framework for building pipelines (Section 2.2). Then we review the current black box model serving approaches in Section 2.3.

Chapter 3 introduces white box model serving; we first identify a set of limitations of black box approaches in Section 3.1, followed by a set of design principles to address the limitations in Section 3.2.

Chapter 4 describes the PRETZEL system as an implementation of the above principles (Section 4.1 to Section 4.3). Section 4.4 contains a set of experiments validating the PRETZEL performance, while Section 4.5 discusses the limitations of current PRETZEL implementation.

Chapter 5 covers related works of our white box approaches and Chapter 6 concludes this dissertation.

Chapter 2

Background

2.1 Machine Learning Pipelines

Many Machine Learning (ML) frameworks such as Google TensorFlow [22], Facebook PyTorch [19], Scikit-learn [63], Spark MLlib [2], H2O [9], or Microsoft ML.NET [14] allow data scientists to declaratively author model *pipeline*s using high-level APIs (e.g., in Python) for better productivity and easy operationalization. Model pipelines are internally represented as Directed Acyclic Graphs (DAGs) of pre-defined operators [1] comprising *data transformations* (e.g., string tokenization, hashing, etc.) for featurization, and *ML models* (e.g., decision trees, linear models, support vector machines, etc.).

Figure 2.1 shows an example pipeline for text analysis whereby input sentences are classified according to the expressed sentiment. When a prediction request is received, the operators in the pipeline behave as follows:

1. *Tokenizer* extracts tokens (e.g., characters, words) from the input string.

2. *Char* and *Word Ngrams* featurize input tokens by extracting character-level

[1] Note that user-defined code can still be executed through a second order operator accepting arbitrary UDFs.

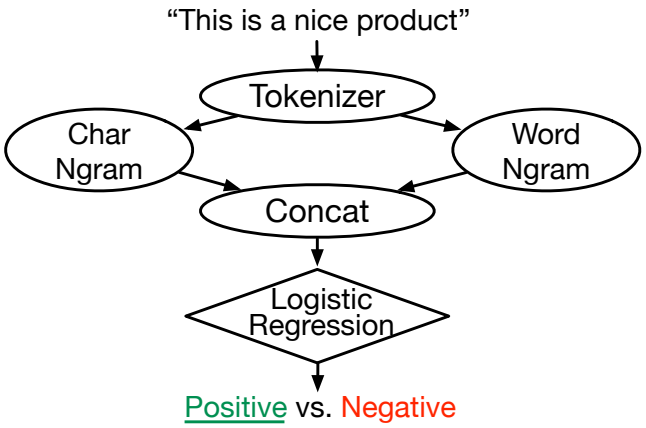


Figure 2.1: An example of ML pipeline: Sentiment Analysis (SA). The pipeline consists of operators for featurization (ellipses), followed by a ML model (diamond).

and word-level n-grams, respectively.

3. *Concat* merges two Ngram vectors and generates a unique feature vector.

4. *Logistic Regression* predictor computes the final score (e.g., 0 or 1 in this example).

Interestingly enough, model pipelines often share similar structures and parameters inasmuch as A/B testing and customer personalization are often used in practice in large scale "intelligent" services; operators could, therefore, be shared between "similar" pipelines. Sharing among pipelines is further justified by how pipelines are authored in practice:

- **Templates**: high-level tools such as ML.NET provide template pipelines for a task based on AutoML techniques. Given task and dataset, ML.NET suggests an appropriate architecture (i.e., operators). Moreover, the operators often come with default values where the probability for the same operators exist between different pipelines.

- **Transfer-learning/Fine-tuning**: to avoid training models from scratch, we often reuse pre-trained pipelines and fine-tune parts of operators.

Operators commonly used for featurizers are frequently reused in other pipelines of the same task (e.g., N-gram operator) with the identical parameters. Even for the pipelines in the different domains, many features are re-used when we apply transfer learning.

2.2 ML.NET

ML.NET [47, 48, 29] is an open-source C# library for writing and running ML pipelines. ML.NET runs on a managed runtime with garbage collection and Just-In-Time (JIT) compilation [2].

ML.NET's main abstraction is called DataView, which borrows ideas from the database community. Similarly to (intensional) database relations, the DataView abstraction provides compositional processing of schematized data, but it is specialized for ML pipelines. In relational databases, the term *view* typically indicates the result of a query on one or more tables (base relations) or views, and is generally immutable [44]. Views have interesting properties that differentiate them from tables and make them appropriate abstractions for ML:

- Views are *composable*; new views are formed by applying transformations (queries) over other views.
- Views are *virtual*; they can be lazily computed on demand from other views or tables without having to materialize any partial results.
- Views are *immutable* and *deterministic*; a view does not contain values but merely computes values from its source views. Therefore, the exact same computation applied over the same input data always produces the same result.

Immutability and deterministic computation enable transparent data caching (for speeding up iterative computations such as ML algorithms) and safe parallel execution. DataView inherits the aforementioned database view properties,

[2] Unmanaged C/C++ code can also be employed to speed up processing when possible.

namely: composability, lazy evaluation, immutability, and deterministic execution.

In ML.NET, pipelines are represented as DAGs of operators, each of which implements the DataView interface, and executes a featurization step or a ML model. Upon pipeline initialization, the operators composing the model pipelines are analyzed and arranged to form a chain of function calls which, at execution time, are JIT-compiled to form a unique function executing the whole DAG on a single call. Operators are able to gracefully and efficiently handle high-dimensional and large datasets thanks to *cursoring*, which resembles the well-known iterator model of databases [43]: within the execution chain, inputs are pulled through each operator to produce intermediate vectors that are input to the following operators, until a prediction or a trained model is rendered as the final output of the pipeline. We refer readers to [47, 48, 29] for further details on ML.NET.

2.3 Black Box Model Serving

Existing prediction serving systems, such as Clipper [4, 38], TensorFlow Serving [23, 61], Rafiki [75], ML.NET [14], and others [18, 20, 56, 15] aim to minimize the overhead of deploying trained pipelines in production. These systems deploy ML models (in general, and pipelines in particular) as *black boxes*, where the same code is used for both training and inference.

Each pipeline is surfaced externally as a black box function. When a prediction request is issued, the invocation of the function on a pipeline returns the result of the prediction; throughout this execution, inputs are pulled through each operator to produce intermediate results that are input to the following operators. Under this approach, internal pipelines' information and structures are not considered inasmuch as pipelines are opaque executable code accepting some input record(s) and producing a prediction.

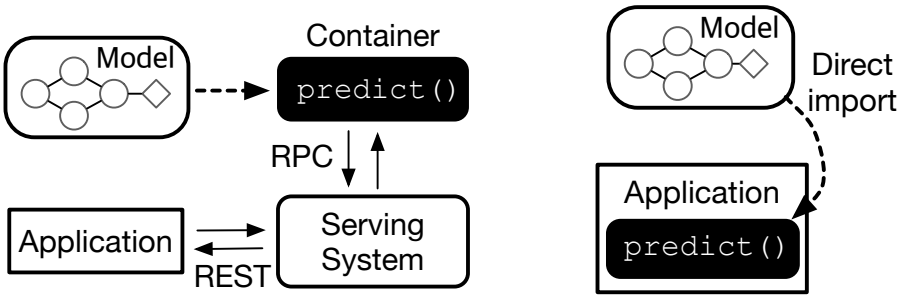
Within the black box approach, there are two ways for a developer to deploy

models, and consequently for an application to request and consume predictions.

The first option (à la Clipper [4], depicted in Figure 2.2(a) and further described in Section 2.3) is to ship models into containers (e.g., Docker [6]) wired with proper Remote Procedure Calls (RPCs) to a Web Server. With this approach, predictions have to go through the network and be rendered on the cloud: low latency or edge scenarios are therefore out of scope.

The second option (Figure 2.2(b) and detailed in Section 2.3) is to integrate the model logic directly into the application (à la ML.NET: the model is a dynamic library the application can link). This approach is suitable for the cloud as well as for edge devices and it unlocks low latency scenarios. However, we still find this approach sub-optimal with respect to customized solutions because it ships the same training pipeline code for prediction. In fact, while using the same code is a great advantage because it removes the need for costly translation work, it implicitly assumes that training and prediction happen in the same regime. However, prediction serving is much more latency-sensitive.

We next provide additional details on each of the two possibilities.



(a) Containerization (b) Embedding model logics

Figure 2.2: Two different ways of deploying models in black box.

Container-based Black Box

Most serving systems in the state of the art [4, 12, 15, 23, 38, 61] aim to minimize the burden of deploying trained pipelines in production by serving

them in containers, where the same code is used for both training and inference [3]. This design allows decoupling models from serving system development and eases the implementation of mechanisms and policies for fault tolerance and scalability. Furthermore, hardware acceleration can be exploited when available.

A typical container-based, model serving system follows the design depicted in Figure 2.2(a): containers are connected to a Serving System (e.g., Clipper) via RPC, and, to score models, applications should contact the Serving System by invoking a Web Server through a REST API. Developers are responsible for setting up the communication between their applications and the Serving System, but in general, this is an easy task as most Serving Systems provide convenient libraries (e.g., Microsoft ML Server [12]).

Implementing model containers for new ML frameworks and integrating them with the Serving System requires a reasonable amount of effort: for example, a graduate student spent a couple of weeks to implement the protocol for integrating an ML.NET container into Clipper.

Embedding Model Logic into Applications

At Microsoft, researchers and developers have encountered the problem of model deployment across a wide spectrum of applications ranging from Bing Ads to Excel, PowerPoint and Windows 10, and running over diverse hardware configurations ranging from desktops to custom hardware (e.g., XBox and IoT devices) and to high-performance servers [1, 7, 13].

To allow such diverse use cases, an ML toolkit deeply embedded into applications should not only satisfy several intrinsic constraints (e.g., scale up or down based on the available main memory and number of cores) but also preserve the benefits commonly associated with model containerization, i.e., (1) it has to capture the full prediction pipeline that takes a test example from a given

[3]Note that TensorFlow Serving [23] is slightly more flexible since users are allowed to split model pipelines and serve them into different containers (called *servables*). However, this process is manual and occurs when building the container image, ignoring the final running environment.

domain (e.g., an email with headers and body) and to produce a prediction that can often be structured and domain-specific (e.g., a collection of likely short responses); and (2) it has to allow to seamlessly carry the complete train-time pipeline into production for model inference. This latter requirement is the keystone for building effective, reproducible pipelines [83].

ML.NET is able to implement all the above desiderata. Once a model is trained in ML.NET, the full training pipeline can be saved and directly surfaced for prediction serving without any external modification.

Figure 2.2(b) depicts the ML.NET solution for black box model deployment and serving: models are integrated into application logic natively and predictions can be served in any OS (Linux, Windows, Android, MacOS) or device supported by the .NET Core framework. This approach removes the overhead of managing containers and implementing RPC functionalities to communicate with the Serving System. In this way, application developers are facilitated for writing applications with ML models inside. Nevertheless, models can still be deployed in the cloud if suggested by the application domain (e.g., because of special hardware requirements).

Chapter 3

White Box Model Serving

3.1 Motivation: Limitations of Black Box Approaches

Although being regarded as a good practice [83], the black box design hides the structure of each served model and prevents the system from controlling and optimizing the pipeline execution. Therefore, under this approach, there is no principled way neither for sharing optimizations between pipelines, nor to improve the end-to-end execution of individual pipelines. More concretely, we observed the following limitations in the current black box prediction serving systems.

3.1.1 Memory Waste

Deploying each model pipeline as a isolated black box (e.g., container) disallows any sharing of resources and runtimes [1] between pipelines, therefore only a few (tens of) models can be deployed per machine. However, ML frameworks such as ML.NET have a known set of operators to start with, and featurizers or models trained over similar datasets have a high likelihood of sharing parameters. For example, transfer learning, A/B testing, and personalized models are common in

[1] One instance of model pipeline in production easily occupies 100s of MB of main memory.

practice; additionally, tools like ML.NET suggest default training configurations to users given a task and a dataset, which leads to many pipelines with similar structure and common objects and parameters. To better illustrate this scenario, we pick a Sentiment Analysis (SA) task with 250 different versions of the pipeline of Figure 2.1 trained by data scientists at Microsoft.

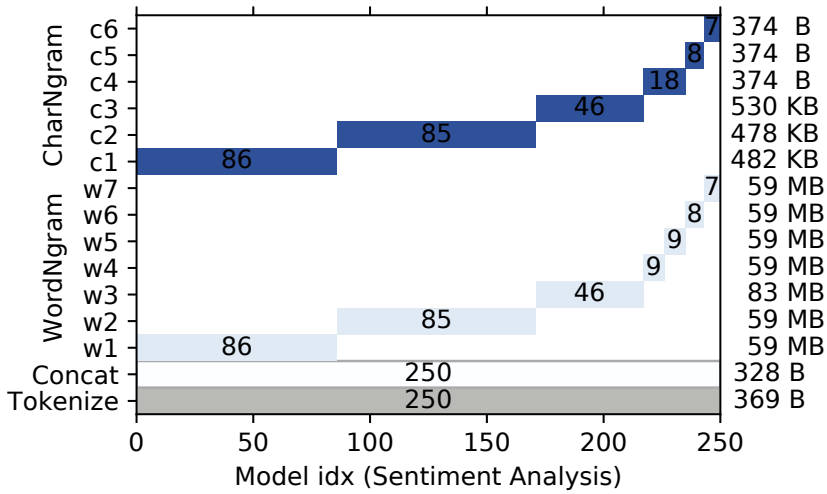


Figure 3.1: How many identical operators can be shared in multiple sentiment analysis pipelines. CharNgram and WordNgram operators have variations that are trained on different hyper-parameters. On the right we report operators sizes.

Figure 3.1 shows how many different (parameterized) operators are used, and how often they are used within the 250 pipelines. While some operators like logistic regression (whose weights fit in ~15MB) are unique to each pipeline and thus not shown in Figure 3.1, many other operators can be shared among pipelines, therefore allowing more aggressive packing of models: Tokenize and Concat are used with the same parameters in all pipelines; Ngram operators have only a handful of versions, where most pipelines use the same version of the operators. This suggests that the memory utilization of the current black box approaches can be largely improved.

3.1.2 Prediction Initialization

ML.NET employs a pull-based execution model that lazily materializes input feature vectors, and tries to reuse existing vectors between intermediate transformations. This largely decreases the memory footprint and the pressure on garbage collection at training time. Conversely, this design forces memory allocation along the data path, thus making latency of predictions sub-optimal and hard to predict.

Furthermore, at prediction time ML.NET deploys pipelines as in the training phase, which requires initialization of function chain call, reflection for type inference and JIT compilation. While this composability conveniently hides complexities and allows changing implementations during training, it is of little use during inference, when a model has a defined structure and its operators are fixed. In general, the above problems result in difficulties in providing strong tail latency guarantees by ML-as-a-service providers.

Figure 3.2 describes this situation, where the performance of *hot* predictions over the 250 sentiment analysis pipelines with memory already allocated and JIT-compiled code is more than two orders of magnitude faster than the worst *cold* case version for the same pipelines.

To drill down more into the problem, we found that 57.4% of the total execution time for a single cold prediction is spent in pipeline analysis and initialization of the function chain, 36.5% in JIT compilation and the remaining is actual computation time.

3.1.3 Infrequent Accesses

In order to meet milliseconds-level latencies [79], model pipelines have to reside in main memory (possibly already warmed-up) but loading and initialization times can easily exceed several seconds since pipelines can have MBs to GBs size on disk. A common practice in production settings is to unload a pipeline if not accessed after a certain period of time (e.g., a few hours). Once evicted,

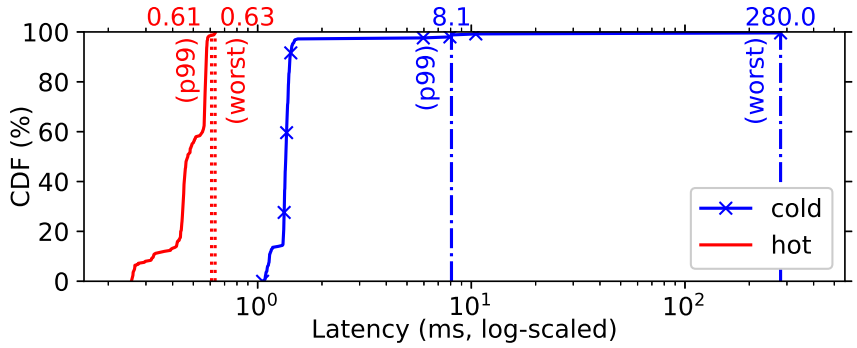


Figure 3.2: CDF of latency of prediction requests of 250 sentiment analysis pipelines. We denote the first prediction as *cold*; the *hot* line is reported as average over 100 predictions after a warm-up period of 10 predictions. We present the 99th percentile and worst-case latency values.

successive accesses will incur a model loading penalty and warming-up, therefore violating Service Level Agreement (SLA).

3.1.4 Operator-at-a-time Model

As previously described, predictions over ML.NET pipelines are computed by pulling records through a sequence of operators, each of them operating over the input vector(s) and producing one or more new vectors. While (as is common practice for in-memory data-intensive systems [60, 74, 30]) some interpretation overheads are eliminated via JIT compilation, operators in ML.NET (and in other tools) are *logical* entities (e.g., logistic regression, tokenizer, one-hot encoder, etc.) with diverse performance characteristics.

Figure 3.3 shows the latency breakdown of one execution of the SA pipeline of Figure 2.1, where the only ML operator (logistic regression) takes two orders-of-magnitude less time with respect to the slowest operator (WordNgram). It is common practice for in-memory data-intensive systems to run operators in parallel (i.e., pipelining) in order to minimize memory accesses for memory-intensive workloads, and to vectorize compute-intensive operators in order to minimize the number of instructions per data item [40, 84].

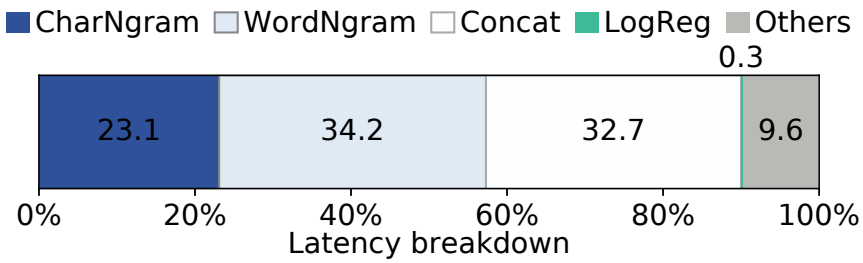


Figure 3.3: Latency breakdown of a sentiment analysis pipeline: each frame represents the relative wall clock time spent on an operator.

ML.NET operator-at-a-time model [84] (as other libraries missing an optimization layer, such as Scikit-learn) is therefore sub-optimal in that computation is organized around logical operators, ignoring how those operators behave together: in the example of the sentiment analysis pipeline at hand, logistic regression is commutative and associative (e.g., dot product between vectors) and can be pipelined with Char and WordNgram, eliminating the need for the Concat operation and the related buffers for intermediate results. As we will see in the following sections, PRETZEL's optimizer is able to detect this situation and generate an execution plan that is several times faster than the ML.NET version of the pipeline.

3.1.5 Coarse Grained Scheduling

Scheduling CPU resources carefully is essential to serve highly concurrent requests and run machines to maximum utilization. Under the black box approach: (1) a thread pool is used to serve multiple concurrent requests to the same model pipeline; (2) for each request, one thread handles the execution of a full pipeline sequentially [2], where one operator is active at each point in time; (3) shared operators/parameters are instantiated and evaluated multiple times (one per container) independently; (4) thread allocation is managed by the OS; and (5) load balancing is achieved *externally* by replicating containers when performance

[2]Certain pipelines allow multi-threaded execution, but here we evaluate only single-threaded ones to estimate the per-thread efficiency.

degradation is observed.

We found this design sub-optimal, especially in heavily skewed scenarios where a small number of popular models are scored more frequently than others: indeed, in this setting the popular models will be replicated (linearly increasing the resources used) whereas containers of less popular pipelines will run underutilized, therefore decreasing the total resource utilization. This problem is currently out-of-scope for black box prediction serving systems because they lack visibility into pipelines execution, and they do not allow models to properly share computational resources.

3.2 Design Principles

We argue that all previously mentioned limitations of the black box approach can be overcome by embracing a *white box approach* allowing to optimize the execution of predictions both horizontally *end-to-end* and vertically *among multiple model pipelines*.

3.2.1 White Box Prediction Serving

The black box approach disallows any sharing of optimizations, resources, and costs between pipelines. By choosing a white box architecture, pipelines can co-exist on the same runtime; unpopular pipelines can be maintained up and warm, while popular pipelines pay the bills. Thorough scheduling of pipelines' components can be managed within the runtime so that optimal allocation decisions can be made for running machines to high utilization. Nevertheless, if a pipeline requires exclusive access to computational or memory resources, a proper reservation-based allocation strategy can be enforced by the scheduler so that container-based execution can be emulated.

3.2.2 End-to-end Optimizations

The operationalization of models for prediction should focus on computation units making optimal decisions on how data are processed and results are

computed, to keep low latency and gracefully degrade with load increase. Such computation units should: (1) avoid memory allocation on the data path; (2) avoid creating separate routines per operator when possible, which are sensitive to branch mis-prediction and poor data locality [60]; and (3) avoid reflection and JIT compilation at prediction time. Optimal computation units can be compiled Ahead-Of-Time (AOT) since pipeline and operator characteristics are known upfront, and often statistics from training are available. The only decision to make at runtime is where to allocate computation units based on available resources and constraints.

3.2.3 Multi-model Optimizations

To take full advantage of the fact that pipelines often use similar operators and parameters (Figure 3.1), shareable components have to be uniquely stored in memory and reused as much as possible to achieve optimal memory usage. Similarly, execution units should be shared at runtime and resources properly pooled and managed, so that multiple prediction requests can be evaluated concurrently. Partial results, for example, outputs of featurization steps, can be saved and re-used among multiple similar pipelines.

Chapter 4

PRETZEL: A White Box ML Inference System

4.1 System Overview

Following the guidelines explained in the previous sections, we implemented PRETZEL, a novel white box system for cloud-based inference of model pipelines. PRETZEL views models as database queries and employs database techniques to optimize pipelines and improve end-to-end performance (Section 4.2.2). The problem of optimizing co-located pipelines is cast as a multi-query optimization and techniques such as view materialization (Section 4.3.1) are employed to speed up pipeline execution. Memory and CPU resources are shared in the form of vector and thread pools, such that overheads for instantiating memory and threads are paid upfront at initialization time.

PRETZEL is organized into several components. A *data-flow-style language-integrated API* called Flour (Section 4.2.1) with related *compiler* and *optimizer* called Oven (Section 4.2.2) are used in concert to convert ML.NET pipelines into *model plans*. An Object Store (Section 4.2.3) saves and shares parameters among plans. A Runtime (Section 4.3.1) manages compiled plans and their execution,

while a Scheduler (Section 4.3.2) manages the dynamic decisions on how to schedule plans based on machine workload. Finally, a FrontEnd is used to submit prediction requests to the system.

In Pretzel, deployment and serving of model pipelines follow a two-phase process. During the *off-line phase* (Section 4.2), ML.NET's pre-trained pipelines are analyzed and compiled into logical and physical stages. In the *on-line phase* (Section 4.3), the system handles inference requests invoking the corresponding stages. Note that only the on-line phase is executed at inference time, whereas the model plans are generated completely off-line.

4.2 Off-line Phase

In the *off-line phase*, ML.NET's pre-trained pipelines are translated into Flour transformations. Oven optimizer re-arranges and fuses transformations into model plans composed of parameterized logical units called *stages*. Each logical stage is then AOT-compiled into physical computation units where memory resources and threads are pooled at runtime. Model plans are registered for prediction serving in the Runtime where physical stages and parameters are shared between pipelines with similar model plans. Figure 4.1 summarizes the process of generating model plans out of ML.NET pipelines.

4.2.1 Flour

The goal of Flour is to provide an intermediate representation between ML frameworks (currently only ML.NET) and Pretzel, that is both easy to target and amenable to optimizations. Once a pipeline is ported into Flour, it can be optimized and compiled (Section 4.2.2) into a model plan before getting fed into Pretzel Runtime for on-line scoring. Flour is a language-integrated API similar to KeystoneML [71], RDDs [81] or LINQ [55] where sequences of *transformations* are chained into DAGs and lazily compiled for execution.

Listing 4.1 shows how the sentiment analysis pipeline of Figure 2.1 can be

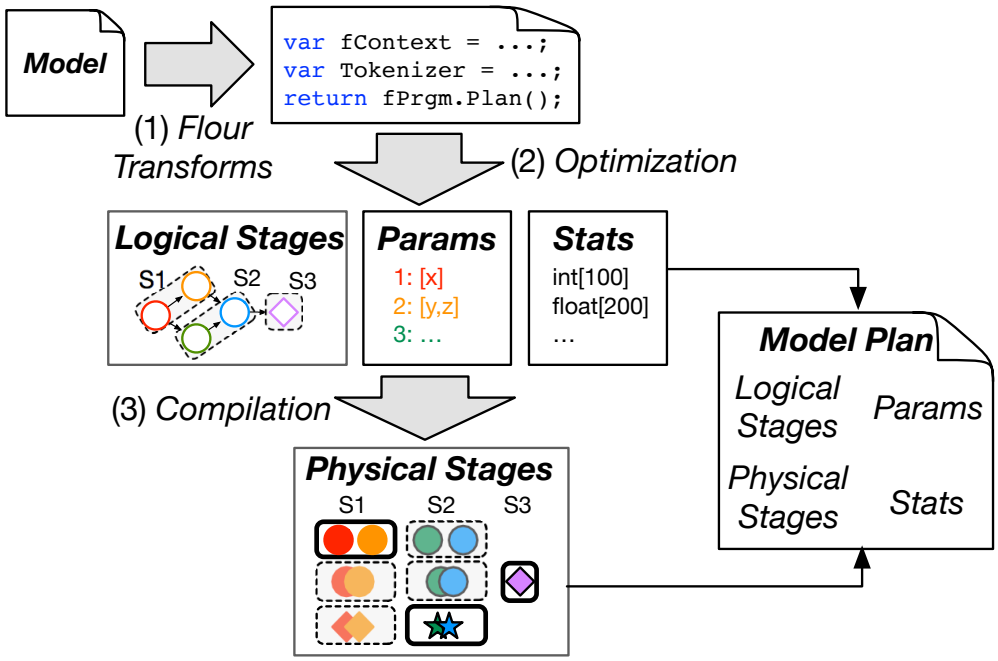


Figure 4.1: Model optimization and compilation in the off-line phase. In (1), a model is translated into a Flour program. (2) Oven Optimizer generates a DAG of logical stages from the program. Additionally, parameters and statistics are extracted. (3) A DAG of physical stages is generated by the Oven Compiler using logical stages, parameters, and statistics. A model plan is the union of all the elements.

expressed in Flour. Flour programs are composed by transformations where a one-to-many mapping exists between ML.NET operators and Flour transformations (i.e., one operator in ML.NET can be mapped to many transformations in Flour).

```
1  var fContext = FlourContext(objectStore, ...);
2  var tTokenizer = fContext.FromText(format=CSV)
3                            .WithSchema<TextReview>()
4                            .Select("Text")
5                            .Tokenize(separator=',');
6
7  var tCNgram = tTokenizer.CharNgram(numCNgrms, ...);
8  var tWNgram = tTokenizer.WordNgram(numWNgrms, ...);
9
10 var fPrgrm = tCNgram.Concat(tWNgram)
11                       .BinaryLinearClassifier(weight);
12
13 return fPrgrm.Plan();
```

Listing 4.1: Flour program for the Sentiment Analysis (SA) pipeline. Parameters are extracted from the original ML.NET pipeline.

Each Flour program starts from a `FlourContext` object wrapping the Object Store. Subsequent method calls define a DAG of transformations, which will end with a call to `Plan` to instantiate the model plan before feeding it into Pretzel Runtime. For example, in line 2 of Listing 4.1 the `FromText(format=CSV)` call is used to specify that the target pipeline accepts as input text in CSV format where fields are comma-separated. Line 3 specifies the schema for the input data, where `TextReview` is a class whose parameters specify the schema fields names, types, and order. The successive call to `Select` in line 4 is used to pick the `Text` column among all the fields, while the call to `Tokenize` in line 5 is used to split the input fields into tokens.

Lines 7 and 8 contain the two branches defining the char-level and word-level n-gram transformations, which are then merged with the `Concat` transform in line 9 before the linear binary classifier of line 10. Both char and word n-gram transformations are parameterized by the number of n-grams and maps translating n-grams into a numerical format (not shown in the Listing).

Additionally, each Flour transformation accepts as input an optional set of statistics gathered from training. These statistics are used by the compiler to generate physical plans more efficiently tailored to the model characteristics. Example statistics are maximum vector size (to define the minimum size of vectors to fetch from the pool at prediction time, as in Section 4.3), dense/sparse representations, etc.

We have instrumented the ML.NET library to collect statistics from training and with the related bindings to the Object Store and Flour to automatically extract Flour programs from pipelines once trained.

4.2.2 Oven

With Oven, our goal is to bring query compilation and optimization techniques of the database systems into ML.NET.

Optimizer

When `Plan` is called on a Flour transformation's reference (e.g., `fPrgrm` in line 13 of Listing 4.1), all transformations leading to it are wrapped and analyzed.

Oven follows the typical rule-based database optimizer design where operator graphs (query plans) are transformed by a set of rules until a fix-point is reached (i.e., the graph does not change after the application of any rule). The goal of Oven Optimizer is to transform an input graph of Flour transformations into a stage graph, where each stage contains one or more transformations. To group transformations into stages we used the Tupleware's hybrid approach [40]: memory-intensive transformations (such as most featurizers) are pipelined together in a single pass over the data. This strategy achieves best data locality because records are likely to reside in CPU L1 caches [51, 60]. Compute-intensive transformations (e.g., vector or matrix multiplications) are executed one-at-a-time so that Single Instruction, Multiple Data (SIMD) vectorization can be exploited, therefore optimizing the number of instructions per record [84, 32]. Transformation classes are annotated (e.g., 1-to-1, 1-to-n, memory-bound, compute-bound, commutative and associative) to ease the optimization process: no dynamic compilation [40] is necessary since the set of operators is fixed and manual annotation is sufficient to generate properly optimized plans [1].

Stages are generated by traversing the Flour transformations graph repeatedly and applying rules when matching conditions are satisfied. Oven Optimizer consists of an extensible number of *rewriting steps*, each of which in turn is composed of a set of rules performing some modification on the input graph.

Each rewriting step is executed sequentially: within each step, the optimizer iterates over its full set of rules until an iteration exists such that the graph is not modified after all rules are evaluated. When a rule is active, the graph

[1] Note that ML.NET does provide a second order operator accepting arbitrary code requiring dynamic compilation. However, this is not supported in our current version of Pretzel.

is traversed (either top-down or bottom-up, based on rule internal behavior; Oven provides graph traversal utilities for both cases) and the rewriting logic is applied if the matching condition is satisfied over the current node.

In its current implementation, the Oven Optimizer is composed of 4 rewriting steps: InputGraphValidatorStep, StageGraphBuilderStep, StageGraphOptimizerStep, and OutputGraphValidatorStep.

1. *InputGraphValidatorStep:* This step comprises three rules, performing schema propagation, schema validation and graph validation. Specifically, the rules propagate schema information from the input to the final transformation in the graph, and validate that (1) each transformation's input schema matches with the transformation semantics (e.g., a WordNgram has a string type as input schema, or a linear learner has a vector of floats as input), and (2) the transformation graph is well-formed (e.g., a final predictor exists).

2. *StageGraphBuilderStep:* It contains two rules that rewrite the graph of (now schematized) Flour transformations into a stage graph. Starting with a valid transformation graph, the rules in this step traverse the graph until a pipeline-breaking transformation is found, i.e., a Concat or an n-to-1 transformation such as an aggregate used for normalization (e.g., L2 norm). These transformations, in fact, require data to be fully scanned or materialized in memory before the next transformation can be executed. For example, operations following a Concat require the full feature vector to be available, or a Normalizer requires the L2 norm of the complete vector. The output of the `StageGraphBuilderStep` is therefore a stage graph, where each stage internally contains one or more transformations. Dependencies between stages are created as an aggregation of the dependencies between the internal transformations. By leveraging the stage graph, PRETZEL is able to considerably decrease the number of vectors (and as a consequence

the memory usage) with respect to the operator-at-a-time strategy of ML.NET.

3. *StageGraphOptimizerStep:* This step involves 9 rules that rewrite the graph in order to produce an optimal logical plan. The most important rules in this step rewrite the stage graph by (1) removing unnecessary branches (i.e., common sub-expression elimination); (2) merging stages containing equal transformations, which are often generated by traversing graphs with branches; (3) inlining stages that contain only one transform; (4) pushing linear models through Concat operations; and (5) removal of unnecessary stages (e.g., when linear models are pushed through Concat operations, the latter stage can be removed if not containing any other additional transformation).

4. *OutputGraphValidatorStep:* This last step is composed of 6 rules. These rules are used to generate each stage's schema out of the schemas of the single internal transformations. Stage schema information will be used at runtime to request properly typed vectors. Additionally, some training statistics are applied at this step: transformations are labeled as sparse or dense, and dense compute-bound operations are labeled as vectorizable. A final validation check is run to ensure that the stage graph is well-formed.

In the example sentiment analysis pipeline of Figure 2.1, Oven is able to recognize that the Linear Regression can be pushed into CharNgram and WordNgram, therefore bypassing the execution of Concat. Additionally, Tokenizer can be reused between CharNgram and WordNgram, therefore it will be pipelined with CharNgram (in one stage) and a dependency between CharNgram and WordNgram (in another stage) will be created. The final plan will therefore be composed of 2 stages, versus the initial 4 operators (and vectors) of ML.NET.

Model Plan Compiler

Model plans have two DAGs: a DAG of *logical stages*, and a DAG of *physical stages*. Logical stages are an abstraction of the results of the Oven Optimizer; physical stages contain the actual code that will be executed by the PRETZEL Runtime. For each given DAG, there is a 1-to-n mapping between logical to physical stages so that a logical stage can represent the execution code of different physical implementations. A physical implementation is selected based on the parameters characterizing a logical stage and available statistics.

Plan compilation is a two-step process. After the stage DAG is generated by the Oven Optimizer, the Model Plan Compiler (MPC) maps each stage into its logical representation containing all the parameters for the transformations composing the original stage generated by the optimizer. Parameters are saved for reuse in the Object Store (Section 4.2.3). Once the logical plan is generated, MPC traverses the DAG in topological order and maps each logical stage into a physical implementation. Physical implementations are AOT-compiled, parameterized, lock-free computation units. Each physical stage can be seen as a parametric function that will be dynamically fed at runtime with the proper data vectors and pipeline-specific parameters. This design allows PRETZEL Runtime to share the same physical implementation between multiple pipelines and no memory allocation occurs on the prediction path (more details in Section 4.3.1).

Logical plans maintain the mapping between the pipeline-specific parameters saved in the Object Store and the physical stages executing on the Runtime as well as statistics such as maximum vector size (which will be used at runtime to request the proper amount of memory from the pool).

4.2.3 Object Store

The motivation behind Object Store is based on the insights of Figure 3.1: since many DAGs have similar structures, sharing operators' state (parameters) can considerably improve memory footprint, and consequently, the number of

predictions served per machine. An example is language dictionaries used for input text featurization, which are often in common among many models and are relatively large.

The Object Store is populated off-line by MPC: when a Flour program is submitted for planning, new parameters are kept in the Object Store, while parameters that already exist are ignored and the stage information is rewritten to reuse the previously loaded one. Parameters equality is computed by looking at the checksum of the serialized version of the objects.

4.3 On-line Phase

In the *on-line phase*, when an inference request for a registered model plan is received, physical stages are parameterized dynamically with the proper values maintained in the Object Store. The Scheduler is in charge of binding physical stages to shared execution units. Figure 4.2 depicts the overall process of on-line inference.

4.3.1 Runtime

Initialization

Model plans generated by MPC are registered in the PRETZEL Runtime. Upon registration, a unique pipeline ID is generated, and physical stages composing a plan are loaded into a system *catalog*. If two plans use the same physical stage, this is loaded only once in the catalog so that similar plans may share the same physical stages during execution.

When the Runtime starts, a set of vectors and long-running thread pools (called *Executors*) are initialized. Vector pools are allocated per Executor to improve locality [42]; Executors are instead managed by the Scheduler to execute physical stages (Section 4.3.2) or used to manage incoming prediction requests by the FrontEnd. Allocations of vector and thread pools are managed by configuration parameters and allow PRETZEL to decrease the time spent in

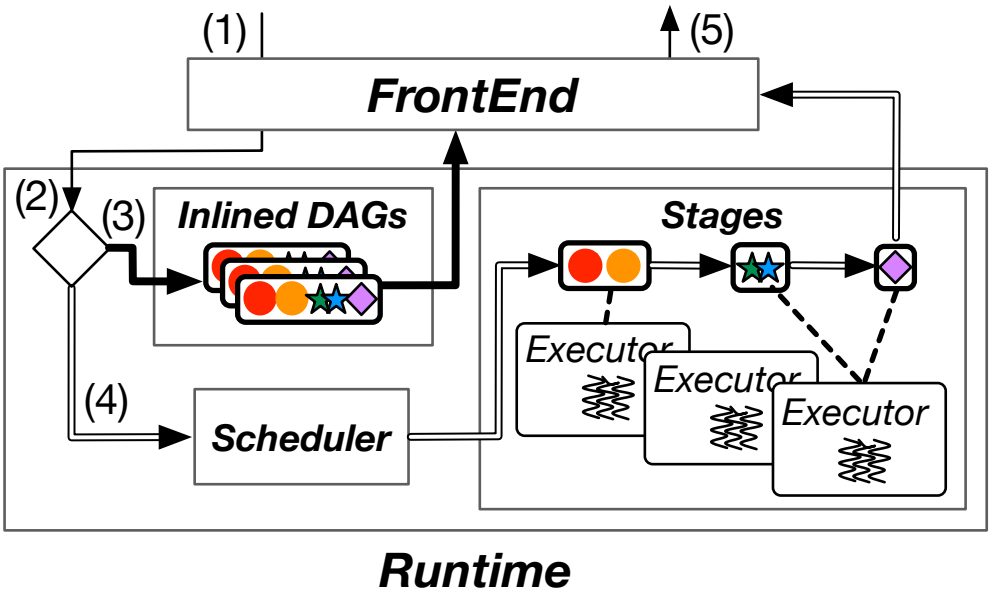


Figure 4.2: Workflow of the on-line phase: (1) When a prediction request is issued, (2) the Runtime determines whether to serve the prediction using (3) the request/response engine or (4) the batch engine. In the latter case, the Scheduler takes care of properly allocating stages over the Executors running concurrently on CPU cores. (5) The FrontEnd returns the result to the Client once all stages are complete.

allocating memory and threads during prediction time.

Execution

Inference requests for the pipelines registered into the system can be submitted through the FrontEnd by specifying the pipeline ID, and a set of input records.

PRETZEL comes with a *request-response engine* and a *batch engine*. The request-response engine is used by single predictions for which latency is the major concern whereby context-switching and scheduling overheads can be costly. Conversely, the batch engine is used when a request contains a batch of records, or when the prediction time is such that scheduling overheads can be considered as negligible (e.g., few hundreds of microseconds). The request-response engine inlines the execution of the prediction within the thread handling the request:

the pipeline physical plan is JIT-compiled into a unique function call and scored. Instead, by using the batch engine requests are forwarded to the Scheduler that decides where to allocate physical stages based on the current runtime and resource status. Currently, whether to use the request-response or batch engine is set through a configuration parameter passed when registering a plan. In the future, we plan to adaptively switch between the two.

Sub-plan Materialization

Similarly to materialized views in database multi-query optimization [44, 35], results of installed physical stages can be reused between different model plans. When plans are loaded in the Runtime, PRETZEL keeps track of physical stages and enables caching of results when a stage with the same parameters is shared by many model plans. Hashing of the input is used to decide whether a result is already available for that stage or not. We implemented a simple Least Recently Used (LRU) strategy on top of the Object Store to evict results when a given memory threshold is met.

4.3.2 Scheduler

In PRETZEL, model plans share resources, thus scheduling plans appropriately is essential to ensure scalability and optimal machine utilization while guaranteeing the performance requirements.

The Scheduler coordinates the execution of multiple stages via a late-binding event-based scheduling mechanism similar to task scheduling in distributed systems [62, 81, 76]. Each core runs an Executor instance whereby all Executors pull work from a shared pair of queues: one *low priority* queue for newly submitted plans, and one *high priority* queue for already started stages.

At runtime, a scheduling event is generated for each stage with a related set of input/output vectors, and routed over a queue (low priority if the stage is the head of a pipeline, high priority otherwise). Two queues with different priorities

are necessary because of memory requirements. Vectors are in fact requested per pipeline (not per stage) and lazily fulfilled when a pipeline's first stage is being evaluated on an Executor. Vectors are then utilized and not re-added to the pool for the full execution of the pipeline. Two priority queues allow started pipelines to be scheduled earlier and therefore return memory quickly.

Reservation-based Scheduling

Upon model plan registration, PRETZEL offers the option to reserve memory or computation resources for exclusive use. Such resources reside on different, pipeline-specific pools, and are not shared among plans, therefore enabling container-like provision of resources. Note however that parameters and physical stage objects remain shared between pipelines even if reservation-based scheduling is requested.

4.3.3 External Optimizations

While the techniques described so far focus mostly on improvements that other prediction serving systems are not able to achieve due to their black box nature, PRETZEL FrontEnd also supports "external" optimizations such as the one provided in Clipper and Rafiki. Specifically, the FrontEnd currently implements prediction results caching with LRU eviction policy and delayed batching whereby inference requests are buffered for a user-specified amount of time and then submitted in batch to the Runtime. These external optimizations are orthogonal to PRETZEL's techniques, so both are applicable in a complementary manner.

4.4 Evaluation

PRETZEL implementation is a mix of C# and C++. In its current version, the system comprises 12.6K LOC (11.3K in C#, 1.3K in C++) and supports about two dozens of ML.NET operators, among which linear models (e.g., linear/logistic/Poisson regression), tree-based models, clustering models (e.g.,

K-Means), Principal Components Analysis (PCA), and several featurizers.

4.4.1 Experimental Setup

The goals of our experimental evaluation are to evaluate how the white box approach performs compared to black box.

Scenarios

We will use the following scenarios to drive our evaluation:

1. *memory*: in the first scenario, we want to show how much memory saving Pretzel's white box approach is able to provide with respect to regular ML.NET and ML.NET boxed into Docker containers managed by Clipper.

2. *latency*: this experiment mimics a request/response pattern [21] such as a personalized web-application requiring minimal latency. In this scenario, we run two different configurations: (1) a micro-benchmark measuring the time required by a system to render a prediction; and (2) an experiment measuring the total end-to-end latency observed by a client submitting a request.

3. *throughput*: this scenario simulates a batch pattern [3] and we use it to assess the throughput of Pretzel compared to ML.NET.

4. *heavy-load*: we finally mix the above experiments and show Pretzel's ability to maintain high throughput and graceful degradation of latency, as load increases. To be realistic, in this scenario we generate skewed load across different pipelines. As for the *latency* experiment, we report first the Pretzel's performance using a micro-benchmark, and then we compare it against the containerized version of ML.NET in an end-to-end setting.

Configuration

All the experiments reported in the paper were carried out on a Windows 10 machine with 2 × 8-core Intel Xeon CPU E5-2620 v4 processors at 2.10GHz with Hyper Threading disabled, and 32GB of RAM. We used .Net Core version 2.0, ML.NET version 0.4, and Clipper version 0.2.

For ML.NET, we use two black box configurations: a non-containerized one (1 ML.NET instance for all pipelines), and a containerized one (1 ML.NET instance for each pipeline) where ML.NET is deployed as Docker containers running on Windows Subsystem for Linux (WSL) and orchestrated by Clipper. We label the former as just ML.NET; the latter as ML.NET + Clipper.

For Pretzel we AOT-compile stages using CrossGen [16]. For the end-to-end experiments comparing Pretzel and ML.NET + Clipper, we use an ASP.Net FrontEnd for Pretzel; the Redis front-end for Clipper.

We run each experiment 3 times and report the median.

Pipelines

Table 4.1 describes the two types of model pipelines we use in the experiments: 250 unique versions of Sentiment Analysis (SA) pipeline, and 250 different pipelines implementing Attendee Count (AC): a regression task used internally to predict how many attendees will join an event.

Pipelines within a category are similar: in particular, pipelines in the SA category benefit from sub-plan materialization, while those in the AC category are more diverse and do not benefit from it. These latter pipelines comprise several ML models forming an ensemble: in the most complex version, we have a dimensionality reduction step executed concurrently with a KMeans clustering, a TreeFeaturizer, and multi-class tree-based classifier, all fed into a final tree (or forest) rendering the prediction.

SA pipelines are trained and scored over Amazon Review dataset [46]; AC ones are trained and scored over an internal record of events.

Table 4.1: Information of pipelines in experiments.

Type	Sentiment Analysis (SA)	Attendee Count (AC)
Task	Classification	Regression
Input	Plain Text (variable length)	Structured Text (fixed: 40 dimensions)
Size	50MB - 100MB (mean: 70MB)	10KB - 20MB (mean: 9MB)
Featurizers	N-gram with dictionaries (∼1M entries)	PCA, KMeans, Ensemble of multiple models
Characteristics	Memory Bound (dictionary lookup)	Compute Bound (multiple featurizers)

4.4.2 Memory

In this experiment, we load all pipelines and report the total memory consumption (pipeline + runtime) per pipeline category. SA pipelines are large and therefore we expect memory consumption (and loading time) to improve considerably within this class, proving that Pretzel's Object Store allows avoiding the cost of loading duplicate objects. Less gain is instead expected for the AC pipelines because of their small size.

Figure 4.3 shows the memory usage for loading all the 250 pipelines in memory, for both categories.

For SA, only Pretzel with Object Store enabled can load all pipelines within the memory limit. While we can load more pipelines in the other configurations and go beyond the 32GB limit, pipelines are swapped to disk and the whole system becomes unstable.

For AC, all configurations are able to load the entire working set, however Pretzel occupies only 164MBs: about 25× less memory than ML.NET and 62× less than ML.NET + Clipper. Given the nature of AC pipelines (i.e., small in size), from Figure 4.3 we can additionally notice the overhead (around 2.5×) of using a container-based black box approach vs regular ML.NET.

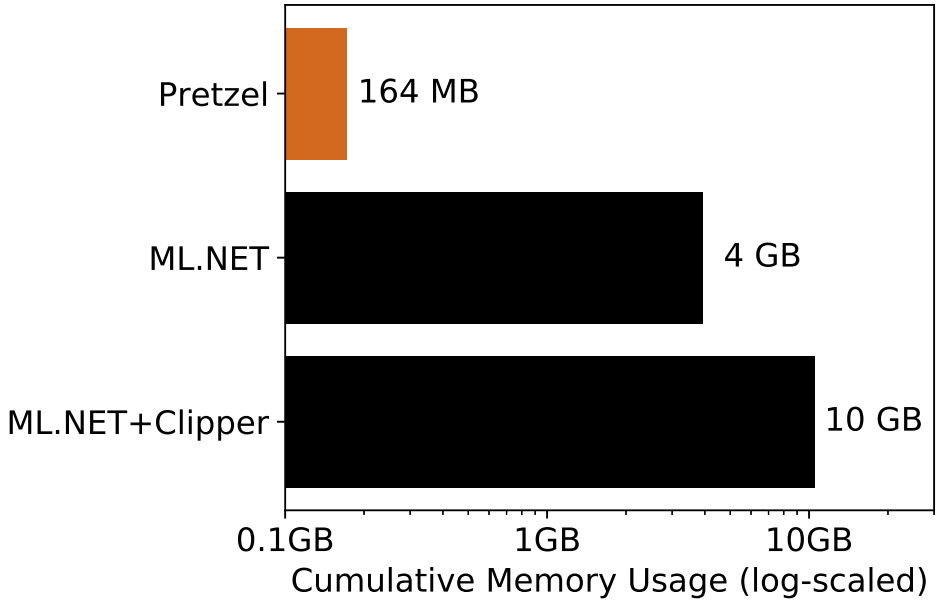


Figure 4.3: Cumulative memory usage (log-scaled) of the pipelines in PRETZEL, ML.NET and ML.NET + Clipper. The horizontal line represents the machine's physical memory (32GB). The memory usage of PRETZEL without Object Store is almost on par with ML.NET.

Keeping track of pipelines' parameters also helps to reduce the time to load pipelines: PRETZEL takes around 2.8 seconds to load 250 AC pipelines while ML.NET takes around 270 seconds. For SA pipelines, PRETZEL takes 37.3 seconds to load all 250 pipelines, while ML.NET fills up the entire memory (32GB) and begins to swap objects after loading 75 pipelines in around 9 minutes.

4.4.3 Latency

In this experiment, we study the latency behavior of PRETZEL in two settings. First, we run a *micro-benchmark* directly measuring the latency of rendering a prediction in PRETZEL. Additionally, we show how PRETZEL's optimizations can improve the latency. Secondly, we report the *end-to-end* latency observed by a remote client submitting a request through HTTP.

Micro-benchmark

Inference requests are submitted sequentially and in isolation for one pipeline at a time. For Pretzel we use the request-response engine over one single core. The comparison between Pretzel and ML.NET for the SA and AC pipelines is reported in Figure 4.4.

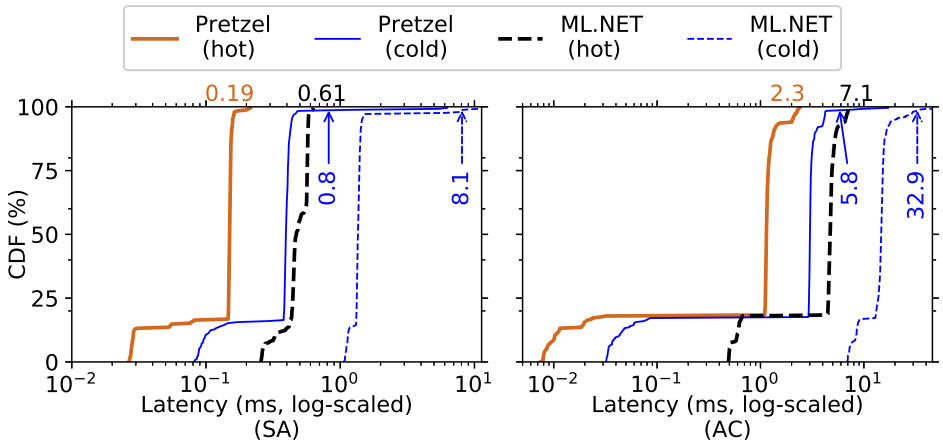


Figure 4.4: Latency comparison between ML.NET and Pretzel. The accompanying blue lines represent the *cold* latency (first execution of the pipelines). On top are the P99 latency values: the hot case is above the horizontal line and the cold case is annotated with an arrow.

We start with studying *hot* and *cold* cases while comparing Pretzel and ML.NET. Specifically, we label as cold the first prediction requested for a pipeline; the successive 10 predictions are then discarded and we report hot numbers as the average of the following 100 predictions.

If we directly compare Pretzel with ML.NET, Pretzel is 3.2× and 3.1× faster than ML.NET in the 99th percentile latency in hot case (denoted by $P99_{hot}$), and about 9.8× and 5.7× in the $P99_{cold}$ case, for SA and AC pipelines, respectively.

If instead we look at the difference between cold and hot cases relative to each system, Pretzel again provides improvements over ML.NET. The $P99_{cold}$ is about 13.3× and 4.6× the $P99_{hot}$ in ML.NET, whereas in Pretzel $P99_{cold}$

is around 4.2× and 2.5× from the $P99_{hot}$ case.

Furthermore, PRETZEL is able to mitigate the long tail latency (worst case) of cold scoring. In SA pipelines, the worst case latency is 460.6× off the $P99_{hot}$ in ML.NET, whereas PRETZEL shows a 33.3× difference. Similarly, in AC pipelines the worst case is 21.2× $P99_{hot}$ for ML.NET, and 7.5× for PRETZEL.

To better understand the effect of PRETZEL's optimizations on latency, we turn on and off some optimizations and compare the performance.

- **AOT compilation:** This option allows PRETZEL to pre-load all stage code into cache, removing the overhead of JIT compilation in the cold cases. Without AOT compilation, latencies of cold predictions increase on average by 1.6× and 4.2× for SA and AC pipelines, respectively.

- **Vector Pooling:** By creating pools of pre-allocated vectors, PRETZEL can minimize the overhead of memory allocation at prediction time. When we do not pool vectors, latencies increase in average by 47.1% for hot and 24.7% for cold, respectively.

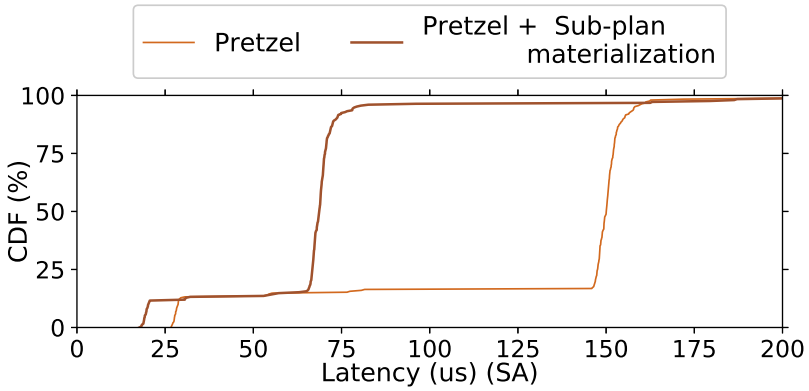


Figure 4.5: Latency of PRETZEL to run SA pipelines with and without sub-plan materialization. Around 80% of SA pipelines show more than 2× speedup. Sub-plan materialization does not apply for AC pipelines.

- **Sub-plan Materialization:** If different pipelines have common featurizers (e.g., SA as shown in Figure 3.1), we can further apply sub-plan

materialization to reduce the latency. Figure 4.5 depicts the effect of sub-plan materialization over prediction latency for hot requests. In general, for the SA pipelines in which sub-plan materialization applies, we can see an average improvement of 2.0×, while no pipeline shows performance deterioration.

End-to-end

In this experiment, we measure the end-to-end latency from a client submitting a prediction request. For PRETZEL, we use the ASP.Net FrontEnd, and we compare against ML.NET + Clipper. The end-to-end latency considers both the prediction latency as well as any additional overhead due to client-server communication. Note that the server-side latency is greater than the latency measured in the micro-benchmark (Figure 4.4). This is because we measure the entire wall clock time in the server, which includes the delay between the server-side Event handler and Executors, in addition to the latency for the actual prediction.

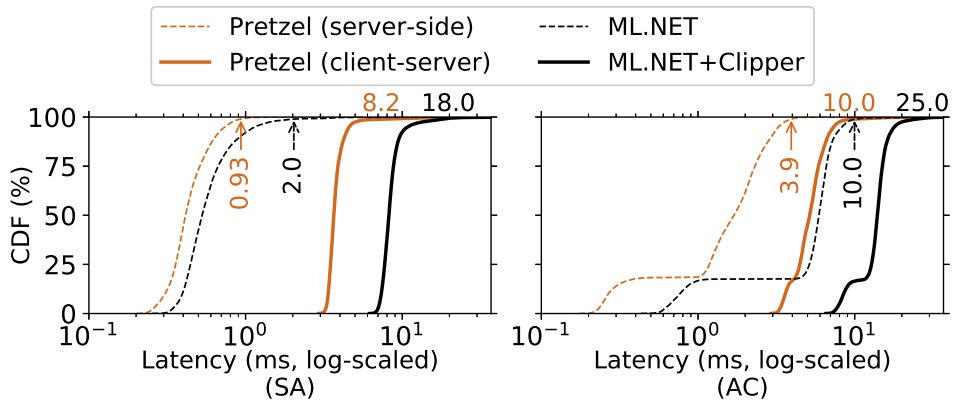


Figure 4.6: The latency comparison between ML.NET + Clipper and PRETZEL with ASP.Net FrontEnd, which involve the overhead of client-server communication. Server-side latency, on the other hand, measures the wall-clock time spent in the server.

As shown in Figure 4.6, the overhead of client-server communication is

similar in both Pretzel and ML.NET: the end-to-end latency compared to the server-side latency is around 9× slower in SA and 2.5× in AC, respectively. If we compare the absolute values, with Pretzel, clients observe a latency of 8.2ms at $P99$ for SA pipelines (vs. 0.93ms $P99$ latency of server-side) and a latency of 10.0ms for AC pipelines (vs. 3.9ms). In contrast, in ML.NET + Clipper, clients observe 18.0ms latency at $P99$ for SA pipelines, and 25.0ms at $P99$ for AC pipelines.

4.4.4 Throughput

In this experiment, we run a micro-benchmark assuming a batch scenario where all 500 pipelines are scored several times. We use an API provided by both Pretzel and ML.NET, where we can execute prediction queries in batches: in this experiment, we fixed the batch size at 1000 queries. We allocate from 2 up to 13 CPU cores to serve requests, while 3 cores are reserved to generate them. The main goal is to measure the maximum number of requests Pretzel and ML.NET can serve per second.

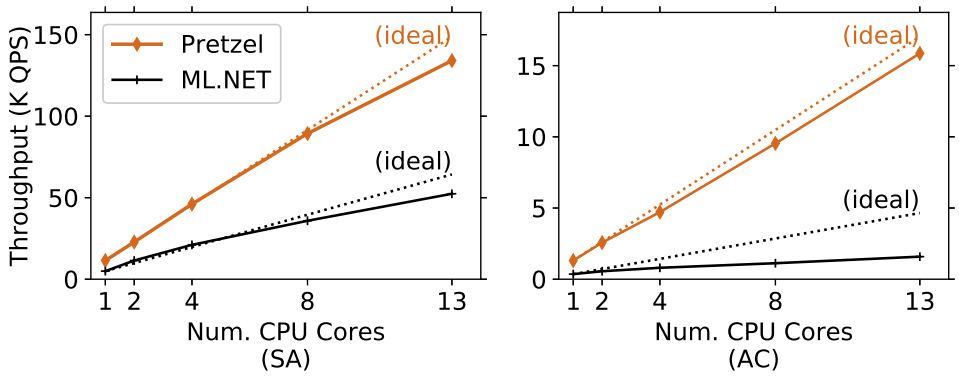


Figure 4.7: The average throughput computed among the 500 pipelines to process one million inputs each. We scale the number of CPU cores on the x-axis and the number of prediction queries to be served per second on the y-axis. Pretzel scales linearly to the number of CPU cores.

Figure 4.7 shows that Pretzel's throughput (queries per second) is up to 2.6× higher than ML.NET for SA pipelines, 10× for AC pipelines. Pretzel's

throughput scales on par with the expected ideal scaling. Instead, ML.NET suffers from higher latency in rendering predictions and from lower scalability when the number of CPU cores increases. This is because each thread has its own internal copy of pipelines whereby cache lines are not shared, thus increasing the pressure on the memory subsystem: indeed, even if the parameters are the same, the pipeline objects are allocated to different memory areas.

4.4.5 Heavy Load

Previous experiments measure latency and throughput exclusively. We focus on a single metric at one experiment, assuming that there is only one type of workload (latency-sensitive vs. throughput-sensitive) and the inputs are ready to be processed in the most efficient way (e.g., large batch for high throughput and single item for low latency). This configuration, however, is too optimistic to represent the complex workload in production.

In this section, we will evaluate how Pretzel handles the prediction requests that are closer to the real-world scenarios. We co-locate pipelines with different characteristics and submit skewed requests across models by following the Zipf distribution ($\alpha = 2$)[2]. We also make requests stochastically over time by following poisson distribution. As in Section 4.4.3, we first show a micro-benchmark, followed by an end-to-end comparison.

Micro-benchmark

We load all 500 pipelines in one Pretzel instance. Among all pipelines, we assume 50% to be “latency-sensitive" and therefore we set a batch size of 1. The remaining 50% pipelines will be requested with 100 queries in a batch. As in the throughput experiment, we use the batch engine with 13 cores to serve requests and 3 cores to generate load.

Figure 4.8 reports the average latency of latency-sensitive pipelines and the

[2]The number of requests to the ith most popular pipelines is proportional to $i^{-\alpha}$, where α is the parameter of the distribution.

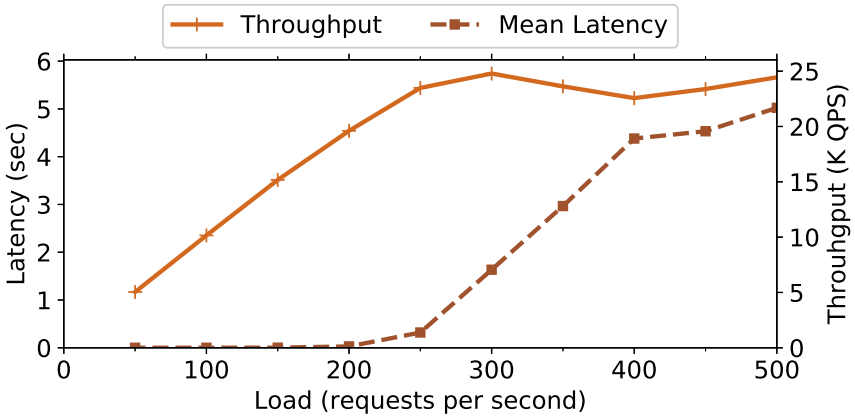


Figure 4.8: Throughput and latency of PRETZEL under the heavy load scenario. We maintain all 500 pipelines in-memory within a PRETZEL instance, and we increase the load by submitting more requests per second. We report latency measurements from latency-sensitive pipelines, and the total system throughput.

total system throughput under different load configurations. As we increase the number of requests, PRETZEL's throughput increases linearly until it stabilizes at about 25k queries per second. Similarly, the average latency of latency-sensitive pipelines gracefully increases linearly with the load.

- **Reservation Scheduling:** If we want to guarantee that the performance of latency-critical pipelines is not degrading excessively even under high load, we can enable reservation scheduling. If we run the previous experiment reserving one core (and related vectors) for one pipeline, this does not encounter any degradation in latency (maximum improvement of 3 orders of magnitude) as the load increases, while maintaining similar system throughput.

End-to-end

In this setup, we periodically send prediction requests to PRETZEL with the ASP.Net FrontEnd and ML.NET + Clipper. We assume all pipelines to be latency-sensitive, thus we set a batch of 1 for each request.

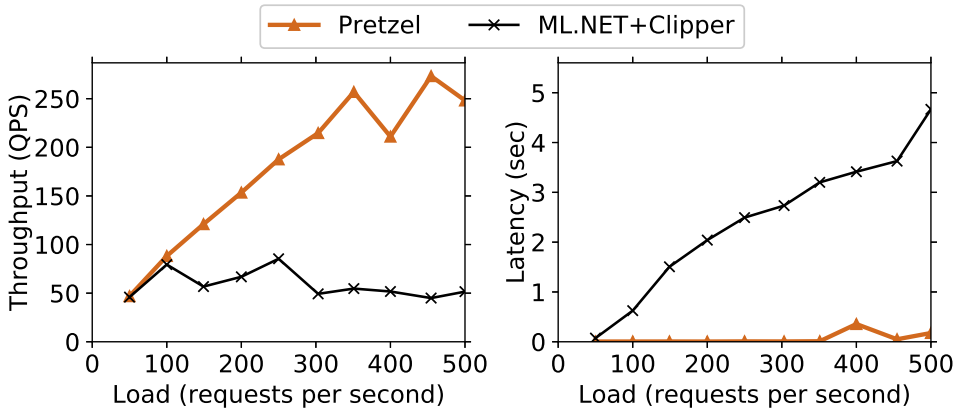


Figure 4.9: Throughput and latency of PRETZEL and ML.NET + Clipper under the end-to-end heavy load scenario. We use 250 AC pipelines to allow both systems to have all pipelines in memory.

As we can see in Figure 4.9, PRETZEL's throughput keeps increasing up to around 300 requests per second. If the load exceeds that point, the throughput and the latency begin to fluctuate. On the other hand, the throughput of ML.NET + Clipper is considerably lower than PRETZEL's and does not scale as the load increases. Also the latency of ML.NET + Clipper is several folds higher than with PRETZEL. The difference is due to the overhead of maintaining hundreds of Docker containers; too many context switches occur across/within containers.

4.5 Limitations

In this section, we will discuss the limitations of the current PRETZEL implementation.

4.5.1 Off-Line Phase

In the off-line phase, PRETZEL has two limitations regarding Flour and Oven design.

1. *Stage Implementation:* PRETZEL currently has several logical and physical

stages classes, one per possible implementation, which makes the system difficult to maintain in the long run. Additionally, different back-ends (e.g., PRETZEL currently supports operators implemented in C# and C++, and experimentally on FPGA [68]) require all specific operator implementations. We are however confident that this limitation will be overcome once code generation of stages will be added (e.g., with hardware-specific templates [53]).

2. *ML Frameworks:* Flour and Oven are currently limited to pipelines authored in ML.NET, and porting models from different frameworks to the white box approach may require non-trivial work. On the long run, our goal is, however, to target unified formats such as ONNX [17]; this will allow us to apply the discussed techniques to models from other ML frameworks as well.

4.5.2 On-Line Phase

In the on-line phase, PRETZEL has the following limitations in its runtime.

1. *Scheduling Overheads:* PRETZEL's fine-grained, stage-based scheduling may introduce additional overheads in contrast to coarse-grained whole pipeline scheduling due to additional buffering and context switching. However, such overheads are related to the system load and therefore controllable by the scheduler.

2. *GC Overheads:* Additionally, we found GC overheads to introduce spikes in latency. Although our implementation tries to minimize the number of objects created at runtime, in practice we found that long-tail latencies are common.

3. *Fault Tolerance:* On white box architectures, failures happening during the execution of a model may jeopardize the whole system. We plan to

add a mechanism to isolate the model failures over the target Executor.

4. *Scalability:* The PRETZEL runtime currently runs on a single-node but we expect in the near future to be able to scale the approach over distributed machines, with automatic scale in/out capabilities.

5. *NUMA awareness:* Finally, PRETZEL runtime currently runs on a single-node. An experimental scheduler adds Non Uniform Memory Access (NUMA) awareness to scheduling policies. We expect this scheduler to bring benefits for models served from large instances (e.g., [8]).

Chapter 5

Related Work

5.1 Prediction Serving

As from the Introduction, current ML prediction systems [4, 38, 23, 61, 18, 36, 20, 56, 75, 15] aim to minimize the cost of deployment and maximize code re-use between training and inference phases [83]. Conversely, PRETZEL casts prediction serving as a database problem and applies end-to-end and multi-query optimizations to maximize performance and resource utilization. Clipper and Rafiki deploy pipelines as Docker containers connected through RPC to a front end. Both systems apply external model-agnostic techniques to achieve better latency, throughput, and accuracy. While we employed similar techniques in the FrontEnd, in PRETZEL we have not yet explored "best effort" techniques such as ensembles, straggler mitigation, and model selection.

TensorFlow Serving deploys pipelines as *Servable*s, which are units of execution scheduling and version management. One Servable is executed as a black box, although users are allowed to split model pipelines and surface them into different Servables, similarly to PRETZEL's stage-based execution. Such optimization is however not automatic. LASER [28] enables large scale training and

inference of logistic regression models, applying specific system optimizations to the problem at hand (i.e., advertising where multiple ad campaigns are run on each user) such as caching partial results and graceful degradation of accuracy.

Both Clipper, Tensorflow Serving, and others [18, 20] allow to serve pipelines exported from different frameworks, and provide techniques to manage model versioning. These features are not currently supported in PRETZEL but can be added with engineering efforts.

Runtimes such as Core ML [5] and Windows ML [27] provide on-device inference engines and accelerators. To our knowledge, only single operator optimizations are enforced (e.g., using target mathematical libraries or hardware), while neither end-to-end nor multi-model optimizations are used.

As PRETZEL, TVM [26, 34] provides a set of logical operators and related physical implementations, backed by an optimizer based on the Halide language [64]. Its optimization is specialized on neural network models and thus does not support featurizers nor "classical" models. Although this limitation can be overcome by translating the classical models into neural network models [77, 78, 57], TVM does not consider multiple models running together and therefore multi-model optimizations cannot be applied.

5.2 Optimizing ML Pipelines

There is a recent interest in the ML community in building languages and optimizations to improve the execution of ML workloads [26, 59, 33, 24, 50]. However, most of them exclusively target Neural Networks and heterogeneous hardware. Nevertheless, we are investigating the possibility to substitute Flour with a custom extension of Tensor Comprehension [73] to express featurization pipelines. This will enable the support for Neural Network featurizers such as word embeddings, as well as code generation capabilities (for heterogeneous devices). We are confident that the set of optimizations implemented in Oven generalizes over different intermediate representations.

Uber's Michelangelo [11] has a Scala DSL that can be compiled into bytecode which is then shipped with the whole model as a zip file for prediction. Similarly, H2O [9] compiles models into Java classes for serving. This is exactly how ML.NET currently works.

Conversely, similar to database query optimizers, PRETZEL rewrites model pipelines both at the logical and at the physical level. KeystoneML [71] provides a high-level API for composing pipelines of operators similarly to Flour, and also features a query optimizer similar to Oven, albeit focused on distributed training. KeystoneML's cost-based optimizer selects the best physical implementation based on runtime statistics (gathered via sampling), while no logical level optimization is provided. Instead, PRETZEL provides end-to-end optimizations by analyzing logical plans [40, 51, 60, 32], while logical-to-physical mappings are decided based on stage parameters and statistics from training. Similarly to the SOFA optimizer [66], we annotate transformations based on logical characteristics. MauveDB [41] uses regression and interpolation models as database views and optimizes them as such. MauveDB models are tightly integrated into the database, thus only a limited class of declaratively definable models is efficiently supported. As PRETZEL, KeystoneML and MauveDB provide sub-plan materialization.

5.3 Scheduling

Both Clipper [4] and Rafiki [75] schedule inference requests based on latency targets and provide adaptive algorithms to maximize throughput and accuracy while minimizing stragglers, for which they both use ensemble models. These techniques are external and orthogonal to the ones provided in PRETZEL.

To our knowledge, no model serving system explored the problem of scheduling requests while sharing resource between models, a problem that PRETZEL addresses with techniques similar to distributed scheduling in cloud computing [62, 80]. Scheduling in white box prediction serving share similarities with

operators scheduling in stream processing systems [31, 72] and web services [76].

Chapter 6

Conclusion

6.1 Summary

Inspired by the growth of ML applications and ML-as-a-service platforms, this dissertation identified how existing systems in black box approaches fall short in key requirements for ML prediction serving, disregarding the optimization of model execution in favor of ease of deployment.

Conversely, we cast ML inference as a database problem, where end-to-end and multi-query optimization strategies can be applied to serving model pipelines in a white box manner. To decrease latency, we have developed an optimizer and compiler framework generating efficient model plans end-to-end. To decrease memory footprint and increase resource utilization and throughput, we allow pipelines to share parameters and physical operators, and defer the problem of inference execution to a scheduler that allows running multiple predictions concurrently on shared resources.

Experiments with production-scale pipelines show the validity of our white box approach in achieving an optimized execution. Pretzel delivers order-of-magnitude improvements on previous approaches over the key performance

metrics: latency, memory footprint and throughput.

6.2 Future Work

In this section, we describe possible extensions to our white box approaches.

A Hybrid Approach of Black Box and White Box

White box approaches accompany with privacy issues because all model information (e.g., operators' types and their parameters) should be exposed to the system. In Pretzel, there is no privacy issue since it originally targets a private cloud environment, where all models are entirely owned by a service provider. This restriction, however, is rigid for Pretzel to be widely adopted.

Instead, we can take a hybrid approach where we combine two methods together. Within a pipeline, there can be some operators that are sensitive to privacy while others are not. For example, pre-trained featurizers are commonly used and thus do not include private-sensitive information. On the other hand, other operators such as ML model consists of sensitive code and data, which users do not want to expose. In this case, we can deploy the sensitive parts to separate black boxes; The other parts can still benefit from the white box optimizations such as graph rewriting, operator sharing, and fine-grained scheduling.

Neural Network Inference Serving

In Pretzel, we have focused on classical machine learning pipelines but we consider extending our scope to support deep neural network (DNN) workload. There exist systems [26, 25, 45, 49, 69, 58] for optimizing the inference of DNN models but there is still plenty of room to study.

Specifically, we can consider optimizations for multiple models that are concurrently served similar to Pretzel's multi-model optimizations. Similar to classical machine learning pipelines, there are neural network operations (e.g., BERT for natural language processing tasks, CNN backbone for vision tasks)

commonly used in many pipelines and we can expect to save resources when we run pipelines on the same server as in PRETZEL.

Sharing computations or parameters across DNN models can be categorized into the following cases:

- Same weights and same inputs (*reusing*): given the same input, we can reuse the intermediate results that are previously computed by other models if the models include the same computation [45, 49]. This is equivalent to sub-plan materialization in PRETZEL (Section 4.3.1).

- Same weights and different inputs (*batching*): we can process multiple requests to increase the throughput. We can apply batching not only when the entire layers have the same weights[38] but also when partial layers consist of the same weights (called "prefix batching")[69]. This is similar to how multiple stages share the same parameters and computation resources in PRETZEL.

- Different weights and same inputs (*layer fusion*): even for the DNN models with different weights, we can merge the layers if the input is the same in order to increase the computational intensity and reduce the kernel launch overhead [58].

The remaining dimension (i.e., different weights and different inputs) has not been explored yet but this optimization will be effective because many DNN models have distinct weights since they are often trained in an end-to-end manner. We believe that combining the above optimizations (e.g., batching[69, 4] and layer fusion[58]) will cover the case and will improve system performance.

Heterogeneous Hardware Accelerators

We can expand the Scheduling layer over heterogeneous hardware, augmented with FPGAs and GPUs for more efficient and predictable computation. Recent work [68] has started exploring the acceleration of ML pipelines on FPGA,

highlighting potential gains of employing such accelerators, whose characteristics are well suited to some ML operators. GPU is also widely used for serving ML models, especially for neural networks. Since each hardware has unique characteristics and data transfer cost occurs across different hardware, the Scheduling layer need to be able to consider these factors for utilizing hardware resources efficiently.

Therefore, in a heterogeneous system, the Scheduling layer can be augmented with FPGA control and decide to schedule some stages on such resources. In this scenario, the Scheduling layer must take into account two major costs. The first cost, which incurs only at the beginning, is programming the FPGA logic with the stage, which is in the order of few seconds; this cost must be accurately accounted, as it may impact the latency of the whole system. The second cost is the data movement cost, as current FPGA chips are accessible only through a PCI-E bus with a latency in the order of microseconds and good performance only in case of sequential memory accesses. Therefore, batching is essential in this scenario to achieve good performance through an FPGA-based accelerator.

Bibliography

[1] AI Platform for Windows Developers. https://blogs.windows.com/buildingapps/2018/03/07/ai-platform-windows-developers

[2] Apache Spark MLlib. https://spark.apache.org/mllib

[3] Batch python API in Microsoft machine learning server.

[4] Clipper. http://clipper.ai

[5] Core ML. https://developer.apple.com/documentation/coreml

[6] Docker. https://www.docker.com

[7] The future of AI marketing: human ingenuity amplified. https://advertise.bingads.microsoft.com/en-us/insights/g/artificial-intelligence-for-marketing

[8] Ec2 large instances and numa. https://forums.aws.amazon.com/thread.jspa?threadID=144982, 2018.

[9] H2O.ai. https://www.h2o.ai

[10] Keras. https://www.tensorflow.org/api_docs/python/tf/keras

[11] Michelangelo. https://eng.uber.com/michelangelo

[12] Microsoft Machine Learning Server. https://docs.microsoft.com/en-us/machine-learning-server

[13] Microsoft Looks To Patent AI For Detecting Video Game Cheaters. https://www.cbinsights.com/research/microsoft-xbox-machine-learning-cheat-detection-gaming-patent

[14] ML.NET. https://dot.net/ml

[15] MXNet Model Server (MMS). https://github.com/awslabs/mxnet-model-server

[16] .Net Core Ahead of Time Compilation with CrossGen. https://github.com/dotnet/coreclr/blob/master/Documentation/building/crossgen.md

[17] Open Neural Network Exchange (ONNX). https://onnx.ai

[18] PredictionIO. https://predictionio.apache.org

[19] PyTorch. https://pytorch.org

[20] Redis-ML. https://github.com/RedisLabsModules/redis-ml

[21] Request response python API in Microsoft machine learning server. https://docs.microsoft.com/en-us/machine-learning-server/operationalize/python/how-to-consume-web-services

[22] TensorFlow. https://www.tensorflow.org

[23] TensorFlow serving. https://www.tensorflow.org/serving

[24] TensorFlow XLA. https://www.tensorflow.org/performance/xla

[25] TensorRT. https://developer.nvidia.com/tensorrt

[26] TVM. https://tvm.ai

[27] Windows ml. `https://docs.microsoft.com/en-us/windows/uwp/machine-learning/overview`

[28] D. Agarwal, B. Long, J. Traupman, D. Xin, and L. Zhang. LASER: A scalable response prediction platform for online advertising. in *Proceedings of Web Search and Data Mining (WSDM)*, 2014.

[29] Z. Ahmed, S. Amizadeh, M. Bilenko, R. Carr, W. -S. Chin, Y. Dekel, X. Dupre, V. Eksarevskiy, E. Erhardt, C. Eseanu, S. Filipi, T. Finley, A. Goswami, M. Hoover, S. Inglis, M. Interlandi, S. Katzenberger, N. Kazmi, G. Krivosheev, P. Luferenko, I. Matantsev, S. Matusevych, S. Moradi, G. Nazirov, J. Ormont, G. Oshri, A. Pagnoni, J. Parmar, P. Roy, S. Shah, M. Z. Siddiqui, M. Weimer, S. Zahirazami, and Y. Zhu Machine Learning at Microsoft with ML.NET in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019

[30] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark SQL: Relational data processing in spark. in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2015.

[31] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas. Operator scheduling in data stream systems. in *The VLDB Journal*, 13(4):333–353, Dec. 2004.

[32] P. A. Boncz, M. Zukowski, and N. Nes. MonetDB/X100: Hyper-pipelining query execution. in *CIDR*, 2005.

[33] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. in *Neural Information Processing Systems (NIPS), Workshop on Machine Learning Systems*, 2015.

[34] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[35] R. Chirkova and J. Yang. Materialized views. in *Foundations and Trends in Databases*, 4(4):295–405, 2012.

[36] D. Crankshaw, P. Bailis, J. E. Gonzalez, H. Li, Z. Zhang, M. J. Franklin, A. Ghodsi, and M. I. Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with Velox. in *CIDR*, 2015.

[37] D. Crankshaw and J. Gonzalez. Prediction-serving systems. in *ACM Queue*, 16(1):70:83–70:97, Feb. 2018.

[38] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica. Clipper: A low-latency online prediction serving system. in *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[39] D. Crankshaw, G.-E. Sela, C. Zumar, X. Mo, J. E. Gonzalez, I. Stoica, and A. Tumanov. InferLine: ML Inference Pipeline Composition Framework, in *CoRR*, 2018.

[40] A. Crotty, A. Galakatos, K. Dursun, T. Kraska, C. Binnig, U. Cetintemel, and S. Zdonik. An architecture for compiling UDF-centric workflows. in *PVLDB*, 8(12):1466–1477, Aug. 2015.

[41] A. Deshpande and S. Madden. MauveDB: Supporting model-based user views in database systems. in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006.

[42] B. Gamsa, O. Krieger, J. Appavoo, and M. Stumm. Tornado: Maximizing locality and concurrency in a shared memory multiprocessor operating system. in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[43] G. Graefe. Volcano: An Extensible and Parallel Query Evaluation System. in *IEEE Transactions on Knowledge and Data Engineering*, February 1994.

[44] A. Y. Halevy. Answering queries using views: A survey. in *The VLDB Journal*, 10(4):270–294, Dec. 2001.

[45] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy. MCDNN: An Approximation-Based Execution Framework for Deep Stream Processing Under Resource Constraints. in *Proceedings of the MobiSys*, 2016.

[46] R. He and J. McAuley. Ups and downs: Modeling the visual evolution of fashion trends with one-class collaborative filtering. in *Proceedings of the World Wide Web Conference (WWW)*, 2016.

[47] M. Interlandi, S. Matusevych, S. Amizadeh, S. Zahirazami, and M. Weimer Machine Learning at Microsoft with ML.NET. in *Neural Information Processing Systems (NIPS), Workshop on Machine learning open source software (MLOSS)*, 2018.

[48] M. Interlandi, S. Matusevych, and M. Weimer ML.NET: Machine Learning Toolkit for Software Developers. in *SysML*, 2019.

[49] A. H. Jiang, D. L.-K. Wong, C. Canel, L. Tang, I. Misra, M. Kaminsky, M. A. Kozuch, P. Pillai, D .G. Andersen and G. R. Ganger Mainstream: Dynamic Stem-Sharing for Multi-Tenant Video Processing. in *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[50] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia. NoScope: Optimizing neural network queries over video at scale. in *PVLDB*, 10(11):1586–1597, Aug. 2017.

[51] A. Kemper, T. Neumann, J. Finis, F. Funke, V. Leis, H. Mühe, T. Mühlbauer, and W. Rödiger. Processing in the hybrid OLTP & OLAP main-memory database system hyper. in *IEEE Data Engineering Bulletin*, 36(2):41–47, 2013.

[52] T. Kornuta, PyTorchPipe: a framework for rapid prototyping of pipelines combining language and vision, in *Neural Information Processing Systems (NeurIPS), Workshop on Machine Learning Systems*, 2019.

[53] K. Krikellas, S. Viglas, and M. Cintra. Generating code for holistic query evaluation. in *ICDE*, 2010.

[54] Y. Lee, A. Scolari, B. -G. Chun, M. D. Santambrogio, M. Weimer, and M. Interlandi. PRETZEL: Opening the black box of Machine Learning Prediction Serving. in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[55] E. Meijer, B. Beckman, and G. Bierman. LINQ: Reconciling object, relations and XML in the .NET framework. in *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2006.

[56] A. N. Modi, C. Y. Koo, C. Y. Foo, C. Mewald, D. M. Baylor, E. Breck, H.-T. Cheng, J. Wilkiewicz, L. Koc, L. Lew, M. A. Zinkevich, M. Wicke, M. Ispir, N. Polyzotis, N. Fiedel, S. E. Haykal, S. Whang, S. Roy, S. Ramesh, V. Jain, X. Zhang, and Z. Haque. TFX: A TensorFlow-based production-scale machine learning platform. in *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2017.

[57] S. Nakandala, G.-I. Yu, M. Weimer, M. Interlandi. in *Neural Information Processing Systems (NeurIPS), Workshop on Machine Learning Systems*, 2019.

[58] D. Narayanan, K. Santhanam, A. Phanishayee, and M. Zaharia. Accelerating Deep Learning Workloads through Efficient Multi-Model Execution. in *Neural Information Processing Systems (NIPS), Workshop on Machine learning Systems*, 2018.

[59] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, K. Duh, M. Faruqui, C. Gan, D. Garrette, Y. Ji, L. Kong, A. Kuncoro, G. Kumar, C. Malaviya, P. Michel, Y. Oda, M. Richardson, N. Saphra, S. Swayamdipta, and P. Yin. DyNet: The dynamic neural network toolkit. in *ArXiv e-prints*, 2017.

[60] T. Neumann. Efficiently compiling efficient query plans for modern hardware. in *PVLDB*, 4(9):539–550, June 2011.

[61] C. Olston, F. Li, J. Harmsen, J. Soyke, K. Gorovoy, L. Lao, N. Fiedel, S. Ramesh, and V. Rajashekhar. Tensorflow-serving: Flexible, high-performance ml serving. in *Neural Information Processing Systems (NIPS), Workshop on ML Systems*, 2017.

[62] K. Ousterhout, P. Wendell, M. Zaharia, and I. Stoica. Sparrow: Distributed, low latency scheduling. in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2013.

[63] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. in *Journal of Machine Learning Research (JMLR)*, 12:2825–2830, Nov. 2011.

[64] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. in *PLDI*, 2013.

[65] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. in *Proceedings of the International Conference on Neural Information Processing Systems (NIPS)*, 2011.

[66] A. Rheinländer, A. Heise, F. Hueske, U. Leser, and F. Naumann. SOFA: an extensible logical optimizer for UDF-heavy data flows. in *Information Systems*, 52:96–125, 2015.

[67] S. Ruder. An overview of gradient descent optimization algorithms. in *CoRR*, 2016.

[68] A. Scolari, Y. Lee, M. Weimer, and M. Interlandi. Towards accelerating generic machine learning prediction pipelines. in *IEEE ICCD*, 2017.

[69] H. Shen, L. Chen, Y. Jin, L. Zhao, B. Kong, M. Philipose, A. Krishnamurthy, and R. Sundaram. Nexus: A GPU Cluster Engine for Accelerating DNN-Based Video Analysis. in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[70] S. Shalev-Shwartz and T. Zhang. Stochastic dual coordinate ascent methods for regularized loss. in *Journal of Machine Learning Research (JMLR)*, 14(1):567–599, Feb. 2013.

[71] E. R. Sparks, S. Venkataraman, T. Kaftan, M. J. Franklin, and B. Recht. KeystoneML: Optimizing pipelines for large-scale advanced analytics. in *Proceedings of the IEEE International Conference on Data Engineering (ICDE)*, 2017.

[72] T. Um, G. Lee, S. Lee, K. Kim, and B.-G. Chun. Scaling up IoT stream processing. in *APSys*, 2017.

[73] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. in *CoRR*, 2018.

[74] S. Wanderman-Milne and N. Li. Runtime code generation in cloudera impala. in *IEEE Data Engineering Bulletin*, 37:31–37, 2014.

[75] W. Wang, S. Wang, J. Gao, M. Zhang, G. Chen, T. K. Ng, and B. C. Ooi. Rafiki: Machine Learning as an Analytics Service System. in *ArXiv e-prints*, Apr. 2018.

[76] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. in *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[77] G.-I. Yu, S. Amizadeh, S. Kim, A. Pagnoni, B.-G. Chun, M. Weimer, M. Interlandi. in *Neural Information Processing Systems (NIPS), Workshop on Machine Learning Systems*, 2018.

[78] G.-I. Yu, S. Amizadeh, S. Kim, A. Pagnoni, B.-G. Chun, M. Weimer, M. Interlandi. Making Classical Machine Learning Pipelines Differentiable: A Neural Translation Approach in *CoRR*, 2019

[79] J.-M. Yun, Y. He, S. Elnikety, and S. Ren. Optimal aggregation policy for reducing tail latency of web search. in *Proceedings of the International ACM SIGIR Conference on Research and Development in Information Retrieval*, 2015.

[80] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica. Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling. in *Proceedings of the EuroSys Conference*, 2010.

[81] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica. Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing. in *Proceedings of the USENIX conference on Networked Systems Design and Implementation (NSDI)*, 2012.

[82] C. Zhang, A. Kumar, and C. Ré. Materialization optimizations for feature selection workloads. in *ACM Transactions on Database Systems (TODS)*, 41(1):2:1–2:32, Feb. 2016.

[83] M. Zinkevich. Rules of machine learning: Best practices for ML engineering. https://developers.google.com/machine-learning/rules-of-ml.

[84] M. Zukowski, P. A. Boncz, N. Nes, and S. Héman. MonetDB/X100 - a DBMS in the CPU cache. in *IEEE Data Engineering Bulletin*, 28(2):17–22, 2005.

요약

머신러닝 추론 시스템은 사용자 서비스를 대상으로 하기 때문에 엄격한 성능 요구사항이 있으며, 구체적으로는 낮은 지연시간 (latency), 높은 처리량 (throughput), 적은 자원 사용을 제공해야 한다. 가장 단순한 방법은 학습된 모델을 컨테이너와 같은 블랙 박스 형태로 배포하여 추론을 수행하는 것이다. 이 방법은 배포 과정을 쉽게 하지만, 적용 가능한 최적화의 범위가 제한되어 있기 때문에 자원을 공유하면서 많은 모델을 수행하고자 하는 환경에서는 최적의 성능을 제공하지 못한다.

본 논문에서는 화이트 방식의 머신러닝 추론 시스템을 제안한다. 이 방식은 모델 전체 (end-to-end)와 다중 모델 (multi-model) 최적화를 가능하게 하는데, 모델 구조를 재정비하여 최적화된 수행 계획을 만들고, 동시에 수행되는 여러 모델이 자원을 효율적으로 공유하도록 한다. 이어서 화이트 박스 방식을 구현한 시스템 Pretzel 에 대해서 소개한다. 실제 프로덕션 환경과 유사한 규모의 모델 파이프라인을 사용한 실험을 통해 기존 블랙 박스 방식 시스템 대비 화이트 방식의 최적화가 지연시간 (latency), 메모리 사용 (memory footprint), 처리량 (throughput) 면에서 큰 성능 향상을 보이는 것을 보였다.

주요어: 머신러닝 추론 시스템, 예측 서빙 시스템, 성능 최적화, 화이트 박스 최적화
학번: 2014-21771